Hyper-AP: Enhancing Associative Processing Through A Full-Stack Optimization

Yue Zha
Department of Electrical and Systems Engineering
University of Pennsylvania
Philadelphia, USA
zhayue@seas.upenn.edu

Jing Li
Department of Electrical and Systems Engineering
University of Pennsylvania
Philadelphia, USA
janeli@seas.upenn.edu

Abstract*—Associative processing (AP) is a promising PIM paradigm that overcomes the von Neumann bottleneck (memory wall) by virtue of a radically different execution model. By decomposing arbitrary computations into a sequence of primitive memory operations (i.e., search and write), AP's execution model supports concurrent SIMD computations *in-situ* in the memory array to eliminate the need for data movement. This execution model also provides a native support for flexible data types and only requires a minimal modification on the existing memory design (low hardware complexity). Despite these advantages, the execution model of AP has two limitations that substantially increase the execution time, i.e., 1) it can only search a single pattern in one search operation and 2) it needs to perform a write operation after each search operation. In this paper, we propose the Highly Performant Associative Processor (*Hyper-AP*) to fully address the aforementioned limitations. The core of *Hyper-AP* is an enhanced execution model that reduces the number of search and write operations needed for computations, thereby reducing the execution time. This execution model is generic and improves the performance for both CMOS-based and RRAM-based AP, but it is more beneficial for the RRAM-based AP due to the substantially reduced write operations. We then provide complete architecture and micro-architecture with several optimizations to efficiently implement *Hyper-AP*. In order to reduce the programming complexity, we also develop a compilation framework so that users can write C-like programs with several constraints to run applications on *Hyper-AP*. Several optimizations have been applied in the compilation process to exploit the unique properties of *Hyper-AP*. Our experimental results show that, compared with the recent work IMP, *Hyper-AP* achieves up to $54\times$/$4.4\times$ better power-/area-efficiency for various representative arithmetic operations. For the evaluated benchmarks, *Hyper-AP* achieves $3.3\times$ speedup and $23.8\times$ energy reduction on average compared with IMP. Our evaluation also confirms that the proposed execution model is more beneficial for the RRAM-based AP than its CMOS-based counterpart.

I. Introduction

Associative processing (AP) is a promising processing-in-memory (PIM) paradigm that tackles the von Neumann bottleneck (memory wall [54]) by virtue of a radically different execution model. In comparison to the von Neumann model where the processor reads input data from the memory and writes computation results back afterwards, the AP's execution model performs SIMD computations *in-situ* in the memory array to eliminate the data movement. By concurrently operating on *every* word where data is stored, AP natively supports arbitrary computations in a *word-parallel* and *bit-serial* manner. A lookup table is generated for a given computation (such as addition), which comprises a list of input patterns and the corresponding computation results. AP then performs a sequence of consecutive search and write operations to 1) compare one input pattern with all stored words in parallel, and 2) write corresponding computation result into the words that match this input pattern. As a result, the number of search/write operations performed for one computation, which determines the execution time, is proportional to the number of input patterns. By performing computations using memory operations (search and write), AP largely reuses existing memory peripheral circuits (e.g. sense amplifier) to minimize the modifications on the memory design. This is different from most existing PIM works [13] [49] [15] [3] [21] that integrate non-conventional memory circuits (e.g., ADC and DAC) into the memory array and increase the hardware complexity. Moreover, due to the bit-serial operation, AP natively supports flexible data types (e.g., arbitrary bit width) and allows programmers to better exploit the benefits of using narrow-precision data types or custom data types [59] [14] [27].

AP has been widely explored in many works [39] [25] [43] [37] [55] [35] because of its PIM capability, superior flexibility and massive parallelism. However, AP performance is fundamentally limited by its execution model, which has two major limitations. At first, the execution model of AP can only search a *single* input pattern in each search operation (*Single-Search-Single-Pattern*). Since the number of input patterns in the lookup table grows *exponentially* with respect to the computation complexity (e.g., data width and type), it needs to perform a large number of search operations for complex computations. Second, this execution model needs to perform a write operation after each search operation (*Single-Search-Single-Write*), resulting in a significant number of write operations considering the first limitation. This substantially increases the execution time, especially for the implementation using emerging non-volatile memory technologies (e.g., RRAM [53]) that have long write latency. Moreover, as the write is performed based on the search result of a *single* input pattern, the second limitation also leads to a low write parallelism, i.e., the ratio between the number of word entries written in parallel and the number of stored word entries is

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00074

low. Therefore, it cannot fully utilize the massive word-level parallelism provided by AP.

In this paper, we propose *Hyper-AP* (**H**igh**l**y **Per**formant AP) to substantially improve the performance of traditional AP through a series of innovations in abstract machine model, execution model, architecture/micro-architecture and compilation framework. First, *Hyper-AP* provides a new abstract machine model and an enhanced execution model to fully address the aforementioned limitations. To address the first limitation (*Single-Search-Single-Pattern*), we leverage the two-bit encoding technique [39] to enhance the search capability of AP's execution model. Rather than simply reuse this encoding technique, we extend this technique by adding several new search keys (Fig. 5c), so that *Hyper-AP* can search *multiple* input patterns in a single search operation (*Single-Search-Multi-Pattern*). This substantially reduces the number of search operations as well as the number of write operations considering the second limitation (*Single-Search-Single-Write*). To address the second limitation, we provide a new abstract machine model, so that the enhanced execution model can accumulate search results and perform a write operation after multiple search operations (*Multi-Search-Single-Write*) to further reduce the number of write operations. Moreover, this also improves the write parallelism as the write is performed based on the accumulated search results of *multiple* input patterns.

We then provide complete architecture and micro-architecture implementations for *Hyper-AP* using RRAM technology. Two additional optimization techniques are applied to further improve the performance of *Hyper-AP*. At first, a *logical-unified-physical-separated* array design is applied to implement the proposed abstract machine model. Compared with the *monolithic* array design used in previous works [56] [39], this array design nearly halves the latency of the write operation to reduce the execution time. Moreover, we also provide a low-cost and low-latency communication interface between memory arrays to reduce the synchronization cost. More details are described in Section IV-B.

Finally, in order to reduce the programming complexity, we develop a compilation framework for *Hyper-AP*, so users can write C-like programs with several constraints (Section V-A) to run applications on *Hyper-AP*. This compilation framework automatically generates and optimizes the lookup tables as well as the data layout for the input programs. Two optimization techniques are developed to further reduce the number of search and write operations. At first, the compilation framework can merge lookup tables to eliminate the write operations for the intermediate computation results. Moreover, immediate operands are embedded into lookup tables through constant propagation to reduce the number of input patterns, thereby reducing the number of search operations. More details on these optimization techniques are presented in Section V-B.

In particular, we made the following major contributions:

1. We propose *Hyper-AP* that provides a new abstract machine model and an enhanced execution model to address the limitations of traditional AP. Specifically, *Hyper-AP* performs SIMD computations in a *Single-Search-Multi-Pattern* and *Multi-Search-Single-Write* manner to reduce the number of search and write operations compared with the *Single-Search-Single-Pattern* and *Single-Search-Single-Write* in the traditional AP.

2. We provide complete architecture and micro-architecture implementations for *Hyper-AP* using the RRAM technology. A *logical-unified-physical-separated* array design is proposed to reduce the write latency, and a low-cost and low-latency communication interface is provided to reduce the synchronization cost.

3. We develop a complete compilation framework to reduce the programming complexity. Users can write C-like programs with several constraints (Section V-A) to run applications on *Hyper-AP*. The compilation framework further improves the computation efficiency by 1) merging lookup tables to eliminate the write operations for the intermediate results, and 2) embedding immediate operands into lookup tables to reduce the number of search operations.

4. We evaluate *Hyper-AP* using both synthetic benchmarks and representative benchmarks from real-world applications. We then provide a comprehensive comparison with the recent work IMP, which is a PIM architecture built on the dot-product capability of RRAM crossbar array [21]. Our evaluation shows that *Hyper-AP* outperforms IMP due to the higher parallelism, the support of flexible data types, and the additional optimization techniques. Specifically, for several representative arithmetic operations, *Hyper-AP* achieves up to $4.1\times$, $54\times$ and $4.4\times$ improvement in throughput, power efficiency and area efficiency respectively compared with IMP. *Hyper-AP* also achieves $3.3\times$ speedup and $23.8\times$ energy reduction on average for representative kernels compared with IMP.

5. Our evaluation also confirms that the proposed execution model can improve the performance for both CMOS-based and RRAM-based AP, but it is more beneficial for RRAM-based AP. This is because the reduction on the number of write operations is larger than that of search operations and RRAM-based AP has highly asymmetric latency of write and search ($T_{write}/T_{search} = 10$) as compared with that in the CMOS-based AP ($T_{write}/T_{search} = 1$).

The rest of the paper is organized as follows. Section II provides background information. Section III describes the abstract machine model and execution model of *Hyper-AP*. Section IV presents the instruction set architecture (ISA) and micro-architecture of *Hyper-AP*. Section V describes the programming interface and the custom compilation framework. Section VI shows the evaluation results. Section VII compares *Hyper-AP* with prior work, followed by Section VIII to conclude the paper.

II. Background and Motivation

A. Brief History

The concept of AP originated in 1950's and had been recognized as one of the two most promising computing models together with the von Neumann model. The development of AP underwent several important stages in history. In 1970's,

Foster laid the foundation for AP by providing a comprehensive overview of the related theory, design, algorithms and applications [20]. Potter and his team carried forward the efforts from early AP prototypes such as STARAN [18] and MPP [7] at Goodyear Aerospace Corporation. They further developed an associative programming model and language [44]. However, the excitement fizzled soon after that, due to the cost limitation.

B. Traditional AP

AP is a parallel machine model that is inherently different from traditional von Neumann model. In von Neumann model, the processor reads input data from the memory and writes computation results back afterwards. Data movement occurs through the system bus and memory hierarchies, all of which impose significant latency and energy penalties. Conversely, AP operates *concurrently* on *every* word where the data is stored, eliminating the need for data movement. Because of the massive word-level parallelism, applications with high data-level parallelism are the suitable workloads for AP.

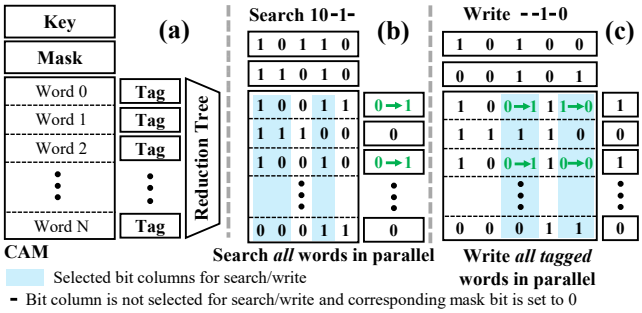

Fig. 1. (a) The abstract machine model of the traditional AP. Examples are drawn to illustrate the (b) search and (c) write operation. The bit selectivity enabled by the mask register is also illustrated. Reduction tree is not drawn in (b, c) for simplicity.

Fig. 1a depicts the abstract machine model of traditional AP. The main building blocks are the content-addressable memory (CAM) array, special-purpose registers (key, mask and tags) and a reduction tree. Specifically, CAM contains an array of bit cells organized in word rows and bit columns. The key register stores the value that can be compared against (Fig. 1b) or written into (Fig. 1c) the CAM words. The mask register defines the active fields for compare/search and write operations, enabling bit selectivity (Fig. 1b, c). Tag registers store comparison results for words and are set to logic '1' upon matches (Fig. 1b). The combination of key and mask can be used to write the selected bit columns of all tagged words in parallel (Fig. 1c). Finally, the reduction tree post-processes the comparison results. Since all word rows perform the same operation (search or write), one word row can be viewed as one SIMD slot.

C. A Simple Motivating Example

AP performs computation in a *bit-serial* and *word-parallel* manner, which is illustrated in Fig. 2 through the 1-bit addition with carry example. In this example, three vectors that have N 1-bit elements (A, B and Cin) are added to produce a result vector (Sum) and a carry vector ($Cout$). These vectors are stored column-wise in the CAM array with one element stored in one row (Fig. 2a). The output vectors (result and carry) are initialized to zero. Then a lookup table is generated for this 1-bit addition (Fig. 2b) and a sequence of search and write operations is carried out to perform this computation (Fig. 2c). Specifically, an input pattern in the lookup table is compared against the contents in the corresponding bit columns of all words *in parallel*, tagging the matches; then computation result (logic '1') is written into the corresponding bit column (Sum or $Cout$) of all tagged words. As shown in Fig. 2c, traditional AP needs 14 operations to complete this 1-bit addition. Other complex operations (e.g., multiplication) can be performed using the same method.

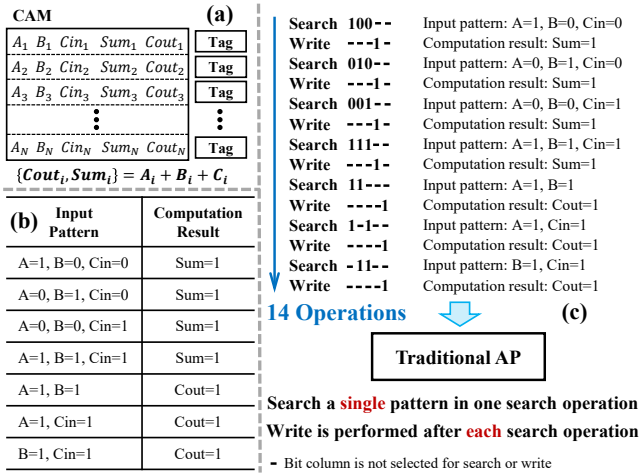

Fig. 2. The 1-bit addition with carry is used as an example to illustrate the SIMD computations performed by traditional AP. The corresponding (a) data layout, (b) lookup table and (c) sequence of search/write operations are drawn.

D. Limitations of Traditional AP

The execution model of traditional AP has two major limitations that could outweigh the benefits obtained from the massive word-level parallelism. At first, this execution model can only search a *single* input pattern in one search operation (*Single-Search-Single-Pattern*), as shown in Fig. 2c. As the number of input patterns in the lookup table exponentially increases with respect to the computation complexity (data width, type, etc.), this limited search capability leads to a large number of search operations (thus a long execution time) for complex computations. Another limitation is that it needs to perform a write operation after each search operation (*Single-Search-Single-Write*, Fig. 2c). This leads to 1) a large number of write operations considering the first limitation, and 2) a low write parallelism (the ratio between the number of tagged words and stored words), since write is performed based on the search result of a *single* input pattern (few tagged words).

E. Ternary Content-Addressable Memory (TCAM)

This subsection briefly describes the 2D2R TCAM architecture that is used to build *Hyper-AP*. Fig. 3a depicts the structure of the crossbar-based TCAM, where one 1D1R cell (one bidirectional diode and one RRAM element) is placed at the intersection of one search line (SL) and one match

line (ML). As shown in Fig. 3b, two adjacent 1D1R cells are paired to store one TCAM bit and two adjacent SLs are paired to represent one bit in the search key. During the search operation, MLs are first precharged to V_{pre} and become floating, then SLs are driven to V_H or V_L based on the input search key and mask. These three voltages are carefully selected ($V_{pre} \approx V_H > V_L$) so that only the difference between V_{pre} and V_L is large enough to turn on the diode. Consequently, a mismatch between the search key and stored word leads to a large discharging current (bottom ML in Fig. 3b), while the match case only has a small discharging current (top ML in Fig. 3b). We use the sense amplifier designed in [39] to sense discharging current and generate search results. 'V/3' scheme [11] is used to write selected RRAMs (Fig. 3c) and diodes can effectively suppress the leakage current in these unselected cells (sneak paths). More details on the 2D2R TCAM architecture can be found in [57] and a chip demonstration can be found in [60].

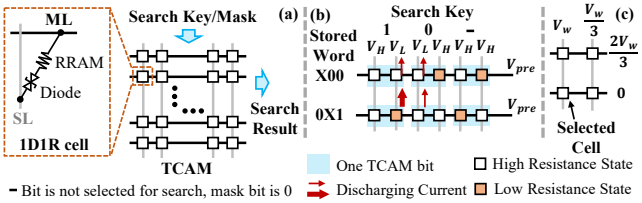

Fig. 3. (a) The structure of the crossbar-based TCAM. (b) A conceptual diagram to illustrate the search operation, where the top ML is the match case, while the bottom ML is the mismatch case. (c) A conceptual diagram to illustrate the write operation.

III. Hyper-AP Execution Model

In this paper, we propose *Hyper-AP* that has a new abstract machine model and an enhanced execution model to address the limitations of traditional AP (Section II-D). The new abstract machine model (Fig. 4a) has three modifications compared with the traditional one (Fig. 1a). At first, the CAM array is replaced by the ternary content-addressable memory (TCAM) array that can store an additional don't care state (denoted by 'X') to match both '0' and '1' input (Fig. 4b). Moreover, the key register is modified to store an additional input state (denoted by 'Z'). This input state only matches 'X' state (Fig. 4c) and is also used to write 'X' into selected bit columns of tagged words (Fig. 4d). Finally, an accumulation unit is added for each tag register to perform logic OR function between the search result and the value stored in the corresponding tag register (Fig. 4c).

Enabled by the abstract machine model, *Hyper-AP* provides an enhanced execution model to fully address the limitations in the traditional execution model. Specifically, enabled by the TCAM and the ternary key register, we extend the two-bit encoding technique (proposed by Li et al. [39]) to enhance the search capability and address the first limitation (*Single-Search-Single-Pattern*). Two-bit encoding technique is proposed to improve the performance of TCAM peripheral circuits by encoding a pair of bits in words (Fig. 5a) and search key (Fig. 5b). Nevertheless, TCAM enhanced with this encoding technique still searches a single pattern in one search operation (Fig. 5b). In this paper, we extend this encoding technique by adding several search keys (Fig. 5c), so that the proposed execution model can search multiple input patterns in a single search operation (*Single-Search-Multi-Pattern*). For instance, a 4-bit search key 1001 will match encoded words X0X1, X00X, 1XX1 and 1X0X if first (last) two bits are paired and encoded. These encoded words are then translated into original words 0001, 0010, 1101 and 1110 based on Fig. 5a, thereby searching four input patterns in one search operation. With the enhanced search capability, *Hyper-AP* only needs 4 search operations to complete the 1-bit addition (Fig. 5d) using the data layout drawn in Fig. 2a and the lookup table shown in Fig. 2b. The number of search operations is $\mathbf{1.8}\times$ fewer than that in the traditional AP (Fig. 2c), and larger reduction can be achieved for more complex computations, e.g., $\mathbf{5.3}\times$ for 32-bit addition. It is also worth pointing out that *Hyper-AP* can store a single bit without encoding, such as Cin in this example. The search key used for the non-encoded bits is the same as that in the traditional AP.

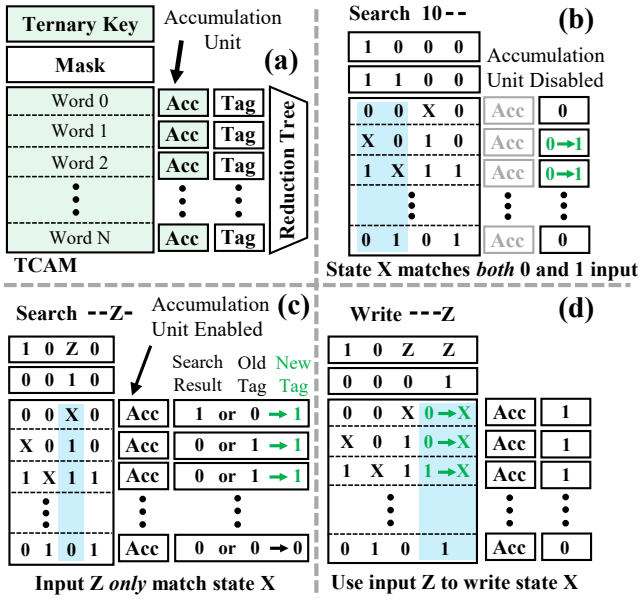

Fig. 4. (a) The abstract machine model of *Hyper-AP* with the modifications highlighted in green. (b) State 'X' matches both '0' and '1' input, and (c) input 'Z' only matches state 'X'. (c) The accumulation unit performs logic OR function. (d) Input 'Z' can also be used to write state 'X'.

To address the second limitation, the enhanced execution model performs one write operation after multiple search operations (*Multi-Search-Single-Write*), which is enabled by the accumulation unit. For instance, the write operation for Sum is only performed *once* after all input patterns related to Sum are searched (Fig. 5d). This reduces the number of write operations by $\mathbf{3.5}\times$ compared with traditional AP (Fig. 2c), and larger reduction can be achieved for more complex computations, e.g., $\mathbf{25.5}\times$ for 32-bit addition. Moreover, as write operations are performed based on the accumulated search results of *multiple* input patterns, the proposed exe-

cution model achieves a higher write parallelism compared with traditional AP. With these improvements, the proposed execution model can complete the 1-bit addition with just 6 operations (Fig. 5d), which is **2.3**× fewer than that in the traditional execution model (Fig. 2c).

(a)

Original Word	Encoded Word
00	X0
01	X1
10	0X
11	1X

(b) Original Search Keys

Search Key	Match Encoded Word	Match Original Word
Z0	X0	00
Z1	X1	01
0Z	0X	10
1Z	1X	11

(c) Additional Search Keys

One search key matches **multiple** words

Search Key	Match Encoded Words	Match Original Words
00	X0, 0X	00, 10
01	X1, 0X	01, 10
10	X0, 1X	00, 11
11	X1, 1X	01, 11
0-	X0, X1, 0X	00, 01, 10
1-	X0, X1, 1X	00, 01, 11
-0	X0, 0X, 1X	00, 10, 11
-1	X1, 0X, 1X	01, 10, 11

X: Match both 0 and 1
Z: Only match X state

– : Bit column is not selected for search/write with mask bit = 0

A and B are paired and encoded, while Cin is stored without encoding

Search 010-- Input patterns: **(1)** A=1, B=0, Cin=0 **(2)** A=0, B=1, Cin=0
Search 101-- Input patterns: **(1)** A=0, B=0, Cin=1 **(2)** A=1, B=1, Cin=1
Write ---1- Computation result: Sum=1
Search -11-- Input patterns: **(1)** A=0, B=1, Cin=1 **(2)** A=1, B=0, Cin=1 **(3)** A=1, B=1, Cin=1
Search 1Z0-- Input pattern: A=1, B=1, Cin=0
Write ----1 Computation result: Cout=1

(d)

6 Operations

Search multiple patterns in one search operation — **Hyper-AP** — **Write is performed after multiple search operations**

Fig. 5. (a) The word encoding and (b) the search keys used in the original two-bit encoding technique [39]. (c) We extend this encoding technique by adding several search keys. (d) With these new search keys, *Hyper-AP* can search *multiple* input patterns in one search operation and only needs 6 operations to complete the 1-bit addition.

IV. Hyper-AP Architecture

In this section, we first describe the instruction set architecture (ISA) that is utilized by our custom compilation framework (Section V-B) to construct arbitrary arithmetic/logic computations with flexible data types. We then present the micro-architecture that implements the abstract machine model of *Hyper-AP* (Fig. 4a).

A. Instruction Set Architecture

The proposed ISA contains 12 instructions that are grouped into three categories (Table I). The Compute instructions are used to perform the associative operations (e.g., search), while the Data Manipulate instructions are used to read/write data registers. The Control instructions are used to set the Group Mask register and synchronize the execution. The functionality of these instructions is described as follows.

1) Search: The Search instruction compares the value stored in the key register with the active field (defined by the mask register) of all stored words in parallel, and the comparison result is stored into the tag registers. In this operation, the accumulation unit will be enabled if the 1-bit <acc> is set to '1', otherwise, the accumulation unit is disabled (<acc>=0). Moreover, the search result will be sent to the Encoder (Fig. 6d) if the 1-bit <encode> is set to '1'.

2) Write: The Write instruction will write the value stored in the key into the TCAM cell selected by the 8-bit column address (<col>) for all tagged words, if the 1-bit <encode> is set to '0'. This instruction can also sequentially write the 2-bit encoded value generated by the Encoder into two selected TCAM cells (<col> and <col>+1) when the <encode> is set to '1'. This instruction will take 12 cycles if it writes a single TCAM cell (1 cycle for decoding address, 1 cycle for setting the key register and 10 cycles for writing one RRAM cell), and it will take 23 cycles if it writes two TCAM cells (1 cycle for decoding address, 2 cycle for setting the key register twice and 20 cycles for writing two RRAM cells).

TABLE I
The Instruction Set Architecture of *Hyper-AP*.

Category	Opcode	Format	Cycles	Length (Byte)
Compute	Search	<acc><encode>	1	1
	Write	<col><encode>	12/23	2
	SetKey	<imm>	1	65
	Count		4	1
	Index		4	1
Data Manipulate	MovR	<dir>	5	1
	ReadR	<addr>	Variable	3
	WriteR	<addr><imm>	Variable	67
	SetTag		1	1
	ReadTag		1	1
Control	Broadcast	<group_mask>	1	2
	Wait	<cycle>	Variable	2

3) SetKey: The SetKey instruction set the key and mask register according to the 512-bit <imm> value. A pair of two bits in <imm> is used to set one bit in the key and mask. Specifically, a bit of mask will be set to '1' and a bit of key will be set to '1'/'0' if the corresponding two bits in <imm> is '01'/'10'. A bit of key (mask) will be set to '1' ('0') if the corresponding two bits are '11' (represent input 'Z'), and will be set to '0' ('0') if they are '00'.

4) Count: The Count instruction performs the population count operation and returns the number of the tagged words.

5) Index: The Index instruction performs the priority encoding operation and returns the index of the first tagged word.

6) MovR: The MovR instruction reads the value in the data register of one PE and stores it into the data register of its adjacent PE. The direction is determined by the 2-bit <dir>. Specifically, this value will be stored into the data register of the PE on the top/left/right/bottom side if <dir> is set to '00'/'01'/'10'/'11'.

7) ReadR: The ReadR instruction reads the value stored in the data register (Fig. 6c) of the PE selected by the 17-

bit <addr>and stores this value into the data buffer of the top-level controller.

8) WriteR: The WriteR instruction stores the 512-bit immediate value (<imm>) into the data register of the PE selected by the 17-bit <addr>.

9) SetTag: The SetTag instruction reads the value stored in the data register of one PE and stores it into the tag registers of the same PE.

10) ReadTag: The ReadTag instruction reads the value stored in the tag registers of one PE and stores it into the data register of the same PE.

11) Broadcast: The Broadcast instruction stores the 8-bit value specified by <group_mask> into the group mask register in the controller.

12) Wait: The Wait instruction stops the execution of one group for a certain number of cycles (specified by the 8-bit <cycle>). This instruction is used by the compilation framework (Section V-B) to synchronize the execution between groups. This synchronization can be realized since the instructions used for computation (Compute category) have deterministic latency and the number of cycles that one group needs to wait can be resolved at the offline compilation time.

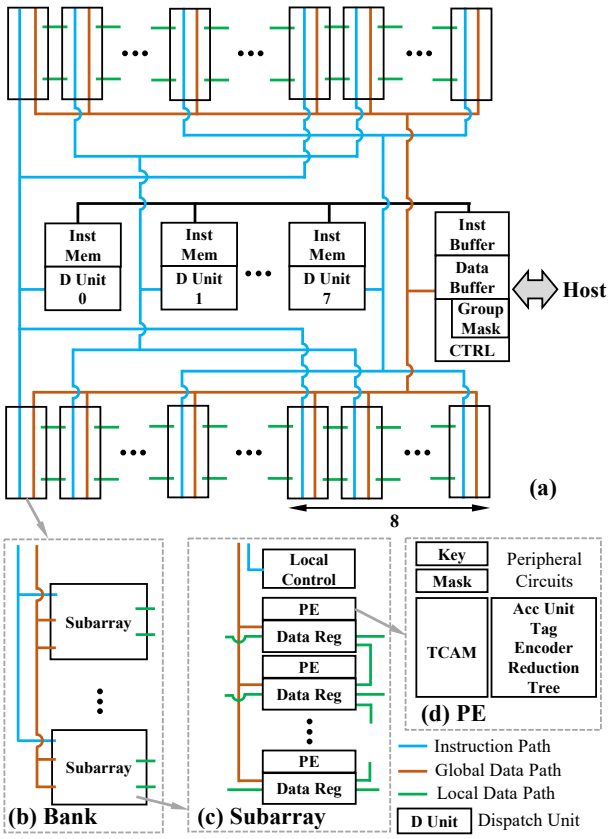

Fig. 6. (a) *Hyper-AP* adopts a hierarchical architecture and comprises a number of banks. (b) Each bank is divided into a set of subarrays, and (c) one subarray further contains several PEs. (d) One PE is a SIMD AP unit to implement the abstract machine model described in Fig. 4a. The micro-architecture of the PE is shown in Fig. 7.

B. Micro-Architecture

The proposed *Hyper-AP* adopts a hierarchical architecture as shown in Fig. 6a. Specifically, *Hyper-AP* comprises a number of banks (Fig. 6b), and each bank is divided into a set of subarrays (Fig. 6c). One subarray further contains multiple PEs (Fig. 6d), and one PE is a SIMD AP unit. The micro-architecture of the PE is shown in Fig. 7. At the top level, banks are grouped to share the instruction memory and dispatch unit, thereby reducing the control overhead. Banks in the same group execute the same instruction (SIMD) to exploit data-level parallelism (DLP), while banks in different groups can execute different instructions to exploit the instruction-level parallelism (ILP). This also allows us to run multiple applications concurrently to exploit task-level parallelism. Therefore, *Hyper-AP* can be viewed as an MIMD architecture. A global data network (yellow data path in Fig. 6a) is included for the data communication between Data Buffer and banks. In addition, one bank is also connected with the adjacent bank (green data path) to realize a *high-bandwidth* and *low-latency* local data communication. As shown in Fig. 6c, a local controller is included in each subarray, which configures the key/mask registers and other peripheral circuits of the PEs based on the received instructions.

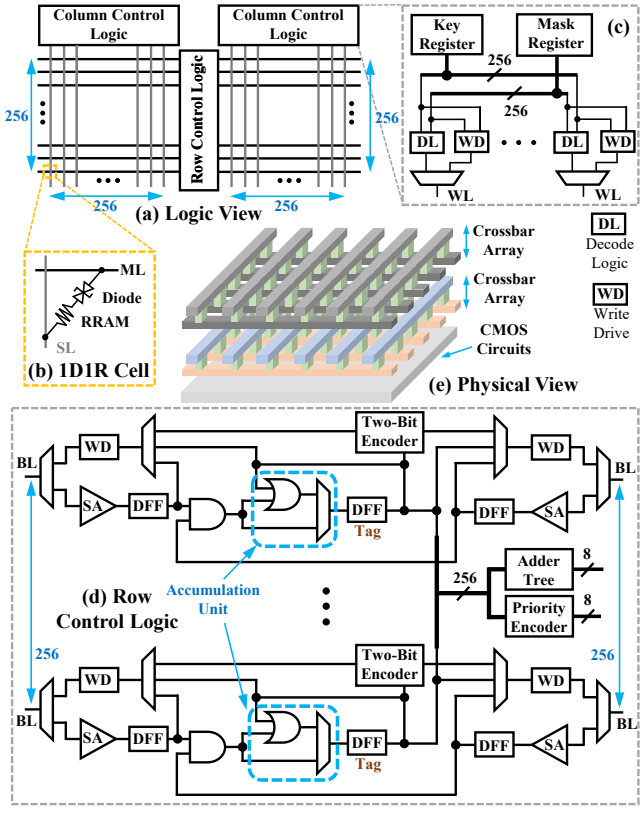

Fig. 7. (a) One PE comprises two RRAM crossbar arrays that implement the TCAM array. Each crossbar array contains 256 × 256 (b) 1D1R cells, (c) a column control logic and (d) a shared row control logic. (e) These two crossbar arrays can be *monolithically* 3D stacked on top of the CMOS circuits without consuming die area.

Fig. 7 depicts the micro-architecture of one PE (Fig. 6d),

which comprises 1) two RRAM crossbar arrays based on 1D1R cell (contains 1 diode [34] and 1 RRAM element [12]) to implement the TCAM array in Fig. 4a, 2) input registers to implement the special purpose key and mask registers in Fig. 4a, 3) output registers to implement the tag registers in Fig. 4a, 4) OR gates and multiplexers to implement the Accumulation Unit in Fig. 4a, 5) an adder tree and priority encoder to implement the reduction tree in Fig. 4a, 6) sense amplifiers (SAs) [39] for search operation and write drivers (WDs) for write operations, 7) two-bit encoders to encode the search results, and 8) column control logic including column decoder and driver. Note that, there is no one-to-one correspondence between 6-8 and the abstract machine model of *Hyper-AP* (Fig. 4a), but they are critical circuits to support AP operations.

The TCAM array design used in this architecture differs from the one used in previous works [37] [56] [25], and has been optimized to reduce the write latency. Specifically, previous works use a *single* RRAM crossbar array to implement the TCAM array and one TCAM bit is stored in two 1D1R cells. Writing a TCAM bit needs to *sequentially* write these two cells as they share the same write circuit. To reduce the write latency, we use *two* RRAM crossbar arrays to implement the TCAM array (Fig. 7a) and the two 1D1R cells of one TCAM bit are in different arrays. Since each array has its own write circuit, this TCAM design can write the two cells of one TCAM bit *in parallel* to halve the write latency.

We employ the same RRAM crossbar array design as the one used in [36] [57], which can be monolithically stacked on top of the CMOS logic without consuming die area (Fig. 7e). To balance the performance, power and cost, the crossbar array size is chosen to be 256×256, and the TCAM array (two crossbar arrays) can store 256 256-bit words (256 SIMD slots). For the search operation, the column decode logic drives the search-lines (SLs) based on the data stored in the key and mask registers and the SAs are activated to sense the match-lines (MLs). Details on the sensing mechanism can be found [39] [57]. The sensing results from the two crossbar arrays are ANDed to generate the final search result. For the associative write operation, the WDs drive the SLs/MLs to the required voltages based on the key/mask/tag values. The 'V/3' write scheme [11] is applied to effectively reduce the leakage current on sneak paths to improve the write performance and reliability. Note that, 1D1R cells in one column (connected to the same SL) can be written in parallel to improve the write throughput.

V. Compilation Framework and System Integration

In this section, we first describe the programming interface of *Hyper-AP* and then present the compilation framework as well as the optimization techniques applied in the compilation flow. Finally, we briefly describe the system integration.

A. Programming Interface

We use a C-like language as the programming interface for *Hyper-AP* to maximally reuse the existing development environment and minimize the efforts for porting programs across different platforms (thus reducing the development cost for *Hyper-AP*). Users write programs to describe the instruction stream that is applied on a single data stream. Our compilation framework then automatically applies this instruction stream onto multiple data streams (Fig. 8). To realize the data alignment for SIMD computations, two constraints are applied on user programs, i.e., 1) loops can be unrolled at the compilation time and the number of loop iterations is the same across all data streams, and 2) pointer chasing operations are not supported. Conditional statements are supported by executing the computations on all branches, as illustrated in Fig. 13b. Nevertheless, it is necessary to minimize the number of conditional statements, since the branch divergence issue causes performance degradation (similar to that in GPGPU [28]). To explore the flexibility provided by AP, this C-like language supports three data types (unsigned int, int and bool) and the integer data types can have an arbitrary bit width (declared by users), as shown in Fig. 8. This C-like language also supports C structure, so that users can define their own custom data types.

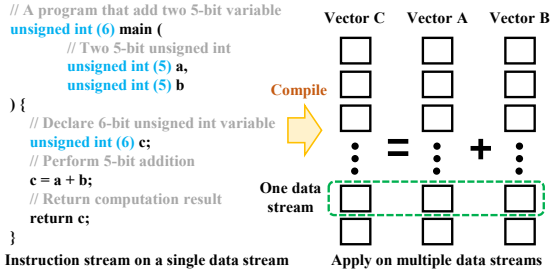

Fig. 8. A C-like language is provided as the programming interface for *Hyper-AP*. Users write programs to describe the instruction stream for a single data stream, and the compilation framework applies this instruction stream on multiple data streams.

B. Compilation Flow

Fig. 9 depicts the compilation framework for *Hyper-AP* that comprises five steps: dataflow graph (DFG) generation, DFG clustering, and-inverter graph (AIG) generation, lookup table generation and code generation. Four optimization techniques have been applied in this compilation flow: 1) function overloading in the AIG generation, 2) two-bits encoding, 3) operation merging and 4) operand embedding in the lookup table generation.

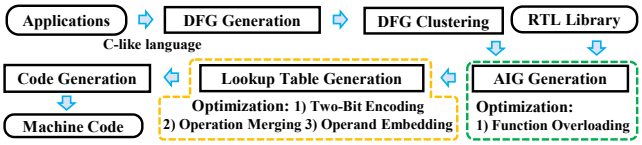

Fig. 9. The compilation flow for *Hyper-AP*.

1) **DFG Generation:** This step parses applications written in the C-like language, performs necessary check to ensure the correctness, and converts it into a DFG. We develop a custom tool for this step.

2) **DFG Clustering:** This step groups the DFG nodes into clusters, and computation in one cluster is performed in one

SIMD slot (i.e., one word row in a PE), as illustrated in Fig. 10. The goal of this step is to minimize the number of edges between clusters, thereby minimizing the number of data copy operations between SIMD slots, which is slow in RRAM-based AP due to the long write latency. We develop a custom tool that uses the heuristic algorithm [42] to cluster the DFG nodes. This clustering algorithm is widely used in the FPGA compilation flow, and we adapt it into our compilation flow by providing a new cost function (Equation 1), i.e., the cost of cluster i is the sum of the cost of its input clusters (cluster j) and the number of input edges.

$$Cost_0[i] = (\sum Cost_0[j]) + N_{input\_edges} \quad (1)$$

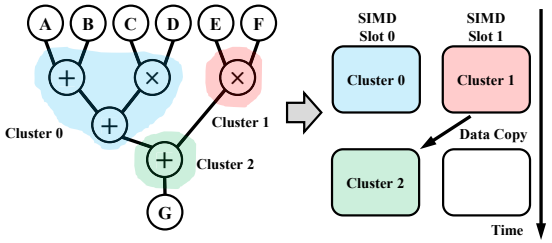

Fig. 10. A conceptual diagram to illustrate the DFG clustering operation. The computation in one cluster is performed in one SIMD slot (one row in a PE), and inter-cluster edges indicate data copy between SIMD slots.

3) **AIG Generation:** For each cluster generated in Step 2, this step replaces the DFG nodes by the corresponding RTL implementations (a netlist of logic gates) provided in the RTL library to generate an AIG for this cluster. This RTL library contains a large number of RTL implementations to support various operations, which are either 1) developed by experts with a hand-optimized performance, or 2) developed by users using high-level programming languages (e.g. C and OpenCL) and high-level synthesis tools [52] [16] [17] to support custom operations. We develop a custom tool for this step to find the appropriate RTL implementation for a given DFG node. This custom tool provides a *function overloading* capability to reduce the programming complexity, which is similar to that in the C++ programming language. Specifically, users can use the same operator (e.g. +) or function name for the different operands (e.g. 8-bit integer or 32-bit unsigned integer). The custom tool finds the appropriate RTL implementation based on the operation type and the data type/precision of the operands.

4) **Lookup Table Generation:** This step groups the AIG nodes into clusters and generates the lookup table for each cluster. This is similar to the technology mapping stage in FPGA compilation flow [41], which generates 6-input lookup tables (LUTs) for a given AIG. Nevertheless, there are two differences between this step and the technology mapping stage. 1) The number of inputs for each cluster is not limited to 6 in this step, but is limited by the search key length, which is 256 in *Hyper-AP*. 2) The optimization goals are different. The technology mapping stage is designed to minimize the critical path length, thereby improving the FPGA operating frequency. While this step is designed to minimize the number of total entries in all lookup tables and the number of writes, thereby reducing the execution time. We develop a custom tool that adapts the heuristic algorithm [42] used in the FPGA technology mapping stage for this step. Two modifications are applied on this algorithm. At first, the limitation on the number of inputs for each cluster is set to 12. We choose this limitation since generating larger clusters only provides a marginal performance improvement (throughput/energy) but substantially increases the compilation time. This also limits the number of bits of the input patterns (thus the search key length) and improves the robustness of the search operation. Moreover, a new cost function is provided (Equation 2), i.e., the cost of cluster i is the sum of the cost of its input clusters (cluster j), the number of patterns in the lookup table and the write latency. Note that, the ratio between the latency of the write operation and the read/search operation is defined as α. By changing the α value, the compilation framework can generate the optimal compilation results for different AP implementations, e.g. CMOS-based AP and RRAM-based AP.

$$Cost_1[i] = (\sum Cost_i[j]) + N_{patterns} + \alpha \quad (2)$$

Three optimization techniques are applied in this step.

a) *Two-Bit Encoding:* The extended two-bit encoding technique (Fig. 5c) is applied to encode the lookup table entries. We notice that different bit pairings lead to different numbers of search operations, as illustrated in Fig. 11. In order to obtain the optimal bit pairing, our custom tool 1) enumerates all possible pairings, 2) counts the number of search operations, and 3) chooses the optimal pairing with the minimum number of searches. Since the number of bits in input patterns is limited (<=12), the search space of this step is small.

b) *Operation Merging:* The clustering operation in the AIG graph can cross the boundary of the DFG nodes, thus, this step can merge multiple operations to eliminate the write of the intermediate results, as illustrated in Fig. 12a.

A	B	C	D	Out
1	0	0	0	1
0	1	0	0	1
1	0	1	1	1
0	1	1	1	1

A and C (B and D) are paired and encoded
Search AC=0Z, BD=Z0 (**Pattern** ABCD=1000)
Search AC=Z0, BD=0Z (**Pattern** ABCD=0100)
Search AC=1Z, BD=Z1 (**Pattern** ABCD=1011)
Search AC=Z1, BD=1Z (**Pattern** ABCD=0111)
Four Searches

A and B (C and D) are paired and encoded
Search AB=01, CD=10 **One Search**

Fig. 11. Different bit pairings lead to different number of search operations.

c) *Operand Embedding:* This step embeds immediate operands into the lookup table through the constant propagation. This optimization reduces the number of search operations (Fig. 12b), thereby reducing the execution time of the corresponding operation. This is a unique feature of *Hyper-AP*, while operations have fixed execution time in the state-of-the-art computing device (e.g., GPU) or previous PIM processors (e.g., IMP [21]).

5) **Code Generation:** This step generates a sequence of SetKey, Search and Write instructions. It then allocates SIMD processors to execute these instructions, and insert the broadcast instruction (Table I) based on the allocation result. Since the instructions used in computation (Compute category in

Table I) have deterministic latency, this step can calculate the execution time for each SIMD processor and insert the wait instruction to synchronize the execution between SIMD processors if necessary. We develop a custom tool for this step.

Fig. 13 gives two examples to show the sequence of search and write operations for 1) arithmetic computations that cannot be completed in a single pass (i.e., just use one large lookup table with many inputs), and 2) the conditional statement.

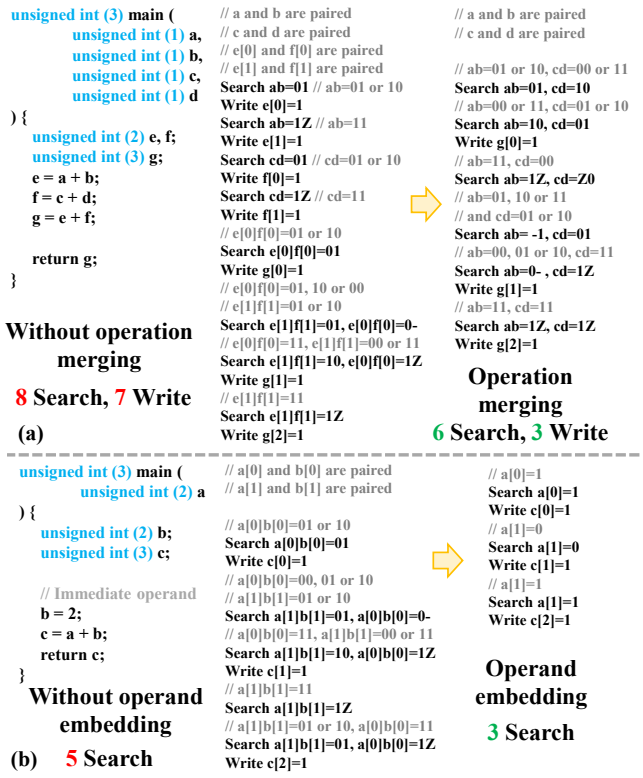

Fig. 12. (a) Merging operations reduces the number of write operations. (b) Embedding immediate operands into lookup tables reduces the number of search operations.

C. System Integration

Hyper-AP can be easily integrated into current systems in the same way as other hardware accelerators (e.g. GPU, FPGA and IMP). For instance, PCIe [2] can be applied to provide a high-performance interconnection for the integration. The host processor loads data into *Hyper-AP* before the computation start, and then streams the instructions into *Hyper-AP* to perform computation.

VI. Evaluation

In this section, we evaluate the performance of RRAM-based *Hyper-AP* on a set of synthetic benchmarks and benchmarks from Rodinia benchmark suite [10]. We also confirm that the proposed execution model (Section III) is more beneficial for the RRAM-based AP by evaluating the performance improvement on CMOS-based and RRAM-based AP. In the following subsections, we first describe the experimental setup, then present the evaluation results.

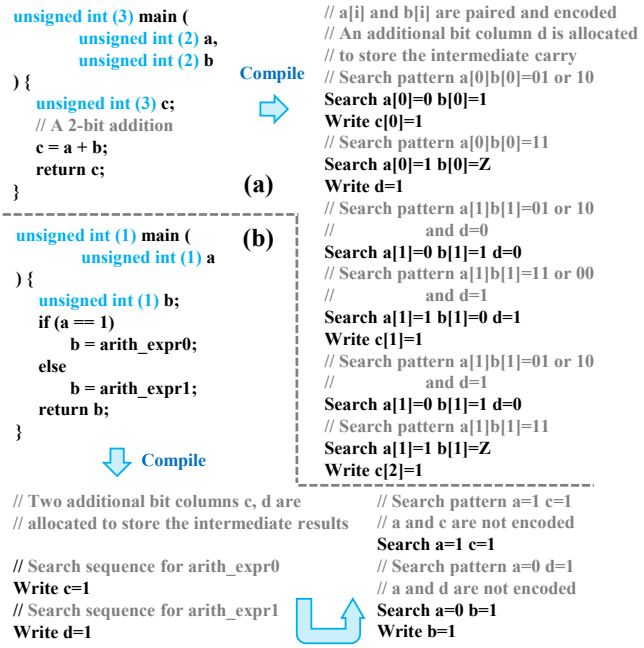

Fig. 13. (a) The sequence of search and write for the 2-bit addition. In this example, the limitation on the number of inputs of the lookup table is 3 (12 in the actual implementation), thus, multiple lookup tables are used to calculate one output bit (i.e., c[1] and c[2]). (b) A simple example to illustrate the handling of conditional statements.

A. Experimental Setup

*1) **Benchmark Selection**:*

Two sets of benchmarks with varying size and complexity are used to evaluate the proposed *Hyper-AP*.

The benchmarks in the first set are relatively small and are synthetically generated to evaluate the computation capability of *Hyper-AP*. Specifically, these benchmarks comprise arithmetic operations that are performed in one SIMD slot (thus no inter-PE communication) to show the peak computing performance (e.g. throughput) achieved by *Hyper-AP*. These benchmarks are also used to show the performance improvement obtained from the 1) flexible data precision support, 2) operation merging, and 3) operand embedding.

The benchmarks in the second set are benchmarks from the Rodinia suite [10] and are used to evaluate both computation and communication performance of *Hyper-AP*. These benchmarks have also been used in the recent work IMP [21], the baseline in our evaluation. To provide a fair comparison, we also convert the floating point numbers in these benchmarks into fixed point numbers, which is the same as that in IMP. The native data set of each benchmark is used, and more details on the benchmark and data set can be found in [21].

*2) **Baseline**:*

The recent work IMP [21] is used as a baseline to provide an apple-to-apple comparison, since it provides the same PIM capability as *Hyper-AP* does, i.e., performing general-purpose computation *in-situ* in the RRAM memory array. To make our evaluation complete, we also include GPU (Nvidia Titan XP) as it is the most popular and commercially available computing

device that provides massive parallelism. Finally, traditional AP is also included in the performance comparison to better illustrate the effectiveness of proposed optimization techniques and the impact of technologies (CMOS and RRAM).

TABLE II
COMPARISON OF GPU, IMP AND *Hyper-AP* PARAMETER.

Parameter	GPU(1-card)	IMP	Hyper-AP
SIMD Slots	3840	2097152	33554432
Frequency	1.58 GHz	20 MHz	1GHz
Area	471mm^2	494mm^2	452 mm^2
TDP	250W	416W	335W
Memory	3MB L2 12GB DRAM	1GB RRAM	1GB RRAM

*3) **Simulation Setup**:*

For *Hyper-AP*, 1) benchmarks are mapped using our compilation framework (Section V-B), 2) the delay and power consumption are obtained from the HSPICE simulation (32nm PTM HP model [5]), and 3) the area is measured based on our custom physical design. More specifically, two Verilog-A modules are created to simulate the behavior of the RRAM device [23] and the diode [34]. The characteristics of the RRAM device are (1) $R_{on}/R_{off} = 20k\Omega/300k\Omega$ and 2) 1.9V@10ns/1.6V@10ns pulse for SET/RESET operation. The turn on voltage of the diode is 0.4V [34]. The capacitance of SL/ML is extracted from the physical design of PE. Note that, since *Hyper-AP* executes instructions in order, instruction latency is deterministic, and the inter-PE communication (local data path in Fig. 6) has fixed latency, the performance can be accurately calculated based on the compilation results. The performance results of GPU and IMP baseline are obtained from the reference [21]. As GPU, IMP and *Hyper-AP* have distinct execution model and architecture, we apply the same evaluation method as IMP [21] to ensure a fair performance comparison. In particular, we have the same assumption as [21] that GPU, IMP and *Hyper-AP* are integrated into the system as standalone accelerators, and the input data has been preloaded into the on-board memory of GPU and the memory array of IMP/*Hyper-AP* before the execution starts.

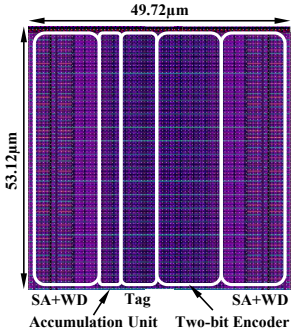

Fig. 14. The physical design of one PE (32nm technology).

B. Configuration Studied

Table II summarizes the important parameters of the three systems compared in this evaluation. *Hyper-AP* provides **16**× more SIMD slots than IMP under the same memory capacity, since one row is one SIMD slot in *Hyper-AP*, while one SIMD slot occupies 16 rows in IMP. Moreover, *Hyper-AP* provides a higher degree of parallelism with a similar die area and less power consumption compared with IMP, since *Hyper-AP* does not need the power-hungry and area-inefficient ADC/DAC. The results of GPU and IMP are obtained from the reference [21]. The area of *Hyper-AP* is obtained from the custom physical design. Fig. 14 shows the physical design of a single PE. Note that PEs in the same subarray can share the key/mask registers, thus, these registers are placed in the local controller instead of in each PE. Moreover, two bits (one in key register and one in mask register) are used to store the three states of one search key bit (0, 1 and -) in traditional AP, and one combination of these two bits are not used. In *Hyper-AP*, we use this combination to store the additonal Z input state without increasing the size of key/mask registers. By monolithically stacking the RRAM crossbar arrays on top of the CMOS circuits, the area of one PE is minimized and is measured as $53.12 \times 49.72 \mu m^2$ at the 32nm CMOS technology node (the same as that in IMP). This physical design also provides key parameters (e.g., parasitic capacitance/resistance) for the circuit simulation to obtain the operating frequency and power consumption.

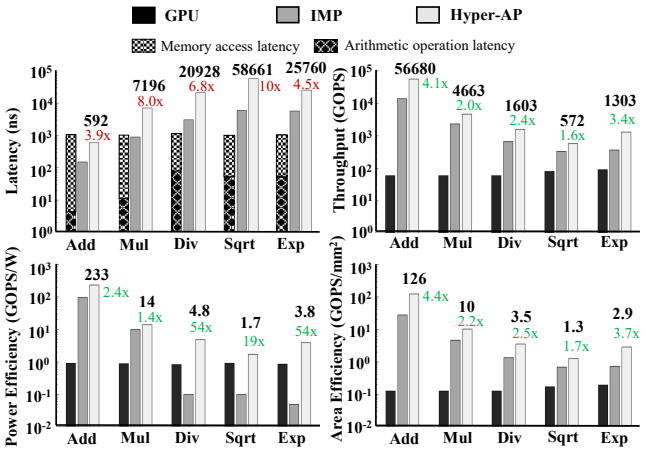

Fig. 15. The latency, throughput, power efficiency and area efficiency results for five representative arithmetic operations using 32-bit unsigned integer. Improvement over IMP is highlighted. The reported benchmark latency of GPU contains the off-chip memory access time and the latency of arithmetic operations (obtained from [4]). In comparison, the benchmark latency of IMP and *Hyper-AP* is equal to the latency of the corresponding arithmetic operations, as IMP and *Hyper-AP* 1) perform computations inside the memory array without accessing the external memory, and 2) can effectively hide the instruction decoding/dispatching latency by overlapping it with computations, while GPU is throughput optimized and not effective to hide the long memory access latency due to the in-order core and limited on-chip memory.

C. Operation Study

In this subsection, the latency, throughput, power efficiency and area efficiency are evaluated using the first synthetic benchmark set. This benchmark set can be further divided into three subgroups that are used to 1) evaluate the performance of a single operation (add, multiple, divide, square root and exponential) on 32-bit integers, 2) evaluate the performance

improvement obtained from the support of flexible data precision, and 3) evaluate the effectiveness of the operation merging and operand embedding techniques (Section V-B).

Fig. 15 presents the evaluation results for the first subgroup. *Hyper-AP* achieves up to $\mathbf{4.1}\times$ improvement in the throughput compared with IMP, since *Hyper-AP* provides a higher degree of parallelism (more SIMD slots), which compensates the relatively longer execution latency in *Hyper-AP* (due to bit-serial operation). *Hyper-AP* also improves the power efficiency by $\mathbf{1.4} \sim \mathbf{2.4}\times$ for addition and multiplication compared with IMP, since *Hyper-AP* does not use the power-hungry ADC/DAC devices for computation. *Hyper-AP* achieves even larger power efficiency improvement (up to $\mathbf{54}\times$) for the complex operations, since *Hyper-AP* supports efficient shift and bit-wise logical operations and can use simple iterative methods [51] [46] [26] to perform these operations, while IMP needs to use look-up table based method. Finally, *Hyper-AP* also improves the area efficiency by $\mathbf{1.7} \sim \mathbf{4.4}\times$ compared with IMP, since *Hyper-AP* uses simple circuits (e.g., sense amplifier) to perform computations, while IMP needs area-inefficient ADC/DAC for computations.

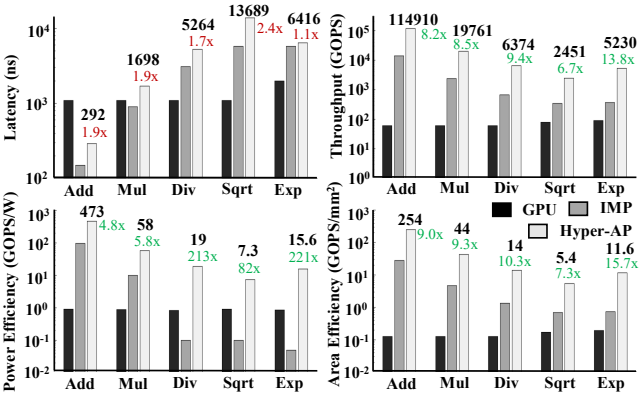

Fig. 16. The latency, throughput, power efficiency and area efficiency results for five representative arithmetic operations using 16-bit unsigned integer. Improvement over IMP is highlighted.

Hyper-AP provides support for a flexible data precision, while IMP only support computations using 32-bit integer. Thus, *Hyper-AP* can achieve higher performance improvement with a reduced data precision. As shown in Fig. 16, for the addition operation, the performance improvement achieved by *Hyper-AP* compared with IMP *linearly* increases with a reduced precision (number of bits), i.e., an additional $\mathbf{2}\times$ improvement is achieved by changing data precision from 32-bit to 16-bit integer. For other complex operations, the performance improvement achieved by *Hyper-AP quadratically* increases with a reduced precision, i.e., an additional $\mathbf{4}\times$ improvement. This is because both the number of iterations and the execution latency of each iteration reduce with a reduced data precision.

Hyper-AP also provides two additional optimizations to improve the performance. At first, multiple operations can be merged to improve the performance since the write operations for the intermediate results are eliminated. As shown in Fig. 17, merging three additions can improve the *Hyper-AP* throughput (76 TOPS) by $\mathbf{1.3}\times$ compared with the non-merged case (56 TOPS in Fig. 15). We note that IMP also provides such operation merging capability and the execution time of the merged operations is nearly the same as that of a single operation, thus, the latency (throughput) gap between *Hyper-AP* and IMP is increased (reduced). However, the operation merging capability of IMP is realized at the cost of higher energy consumption as it requires a higher ADC resolution. On the contrary, by reducing the number of search/write operations, *Hyper-AP* reduces the energy consumption for the merged operations. Therefore, *Hyper-AP* achieves $\mathbf{2.9}\times$ higher power efficiency than IMP, which is $1.2\times$ higher than that of a single operation ($2.4\times$ in Fig. 15).

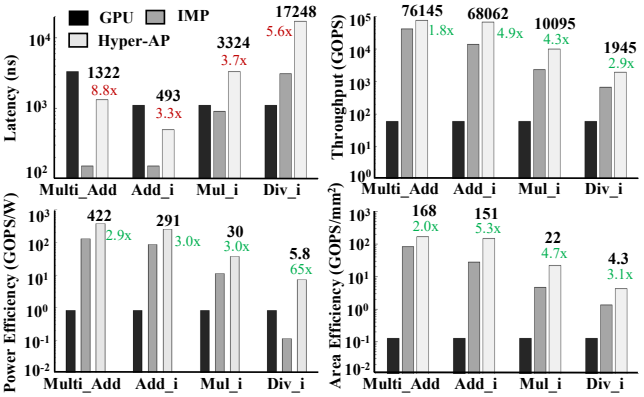

Fig. 17. The latency, throughput, power efficiency, and area efficiency results for 1) three consecutive additions on 32-bit unsigned integer (Multi_Add), and 2) computations with immediate operand (Add_i, Mul_i, Div_i). Improvement over IMP is highlighted.

Moreover, *Hyper-AP* can also improve the performance by embedding the immediate operands into the lookup tables to reduce the number of search operations. Fig. 17 shows the performance results for three operations with immediate operand. On average, *Hyper-AP* can achieve an additional $\mathbf{1.6}\times$ improvement on latency, throughput, power efficiency and area efficiency compared with the results reported in Fig. 15. Note that, square root and exponential are unary operations, and will be computed by the compiler if it has immediate operand. Thus, they are not included in this comparison.

D. Application Study

In this subsection, we use the Rodinia kernels to evaluate the performance of these three systems. As shown in Fig. 18, *Hyper-AP* can achieve $\mathbf{3.3}\times$ speedup on average compared with IMP. The reason for this improvement is two fold. At first, *Hyper-AP* provides $16\times$ more SIMD slots than that in IMP, which allows us to improve the performance by data duplication. Moreover, *Hyper-AP* provides a high-bandwidth and low latency inter-PE communication interface between adjacent PEs (10ns latency and 51.2Gb/s bandwidth). With an optimized data layout, this local data path can substantially reduce the communication cost between SIMD slots in *Hyper-AP*. On the contrary, although IMP also optimizes its data layout, it uses a router-based network for the communication

between SIMD slots, thereby having a relatively higher synchronization cost. We also note that IMP has a relatively higher speedup on Backprop kernel than *Hyper-AP*. This is because IMP natively supports dot-product multiplication, which is heavily used in this kernel. However, this support is realized by using the power-hungry ADC/DAC devices. Thus, IMP has a $\mathbf{23.8}\times$ higher energy consumption on average compared with *Hyper-AP*, as shown in Fig. 18.

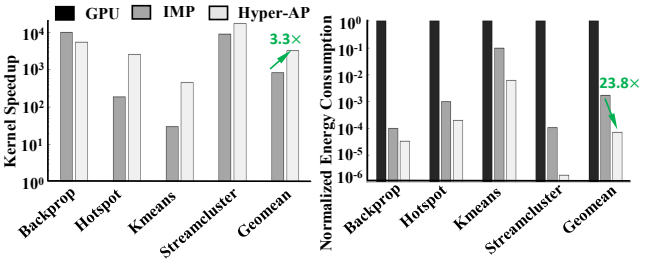

Fig. 18. The speedup (left) and energy consumption (right) results for the Rodinia kernels. The energy consumption of *Hyper-AP* and IMP are normalized to that of GPU.

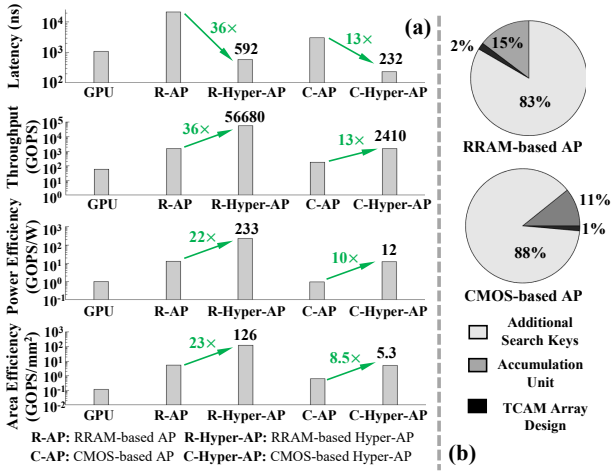

Fig. 19. (a) The performance improvement of the proposed *Hyper-AP* over the traditional AP for both RRAM-based and CMOS-based implementations. (b) The breakdown of the throughput improvement, which mainly comes from the additional search keys (Fig. 5c).

E. Compare With Traditional AP

In this subsection, we use the 32-bit addition as an representative operation to evaluate the performance improvement of the proposed *Hyper-AP* over the traditional AP for both RRAM-based and CMOS-based implementations. Fig. 19a shows that the proposed *Hyper-AP* is more beneficial for the RRAM-based implementation compared with the CMOS-based implementation. Specifically, the RRAM-based *Hyper-AP* can improve the latency/throughput by $\mathbf{36}\times$ compared with the RRAM-based traditional AP. While the CMOS-based *Hyper-AP* can only improve the latency/throughput by $\mathbf{13}\times$ compared with the CMOS-based traditional AP. The similar trend is also observed for other performance metrics (Fig. 19a). The major reason is that the reductions of the search and write operation in *Hyper-AP* are asymmetric, i.e., the number of the search operations is reduced by $5.3\times$, while the number of write operations is reduced by $25.5\times$. As the ratio between the latency of the write operation and search operation in the RRAM-based implementation ($T_{write}/T_{search} = 10$) is much higher than that in the CMOS-based implementation ($T_{write}/T_{search} = 1$), a higher performance improvement can be obtained in the RRAM-based implementation. We also note that although CMOS-based *Hyper-AP* has a better latency (232 ns) compared with RRAM-based Hyper-AP (592 ns), CMOS-based *Hyper-AP* has a lower throughput (2.4 TOPS). This is because, compared with RRAM-based TCAM, CMOS-based TCAM has a much lower storage density, which substantially increases the PE area for the CMOS-based AP (high implementation cost) and reduces the number of SIMD slots. Fig. 19b shows the breakdown of the throughput improvement, which mainly comes from the additional search keys proposed in this paper (Fig. 5c).

VII. Related Work

A. Associative Processing

The recent advances in emerging non-volatile memory have fundamentally lowered the cost barrier of implementing AP. Thus, a number of AP variants have been explored, which are broadly divided into two classes, 1) application-specific AP, represented by CAM or Ternary CAM (TCAM) [39] [29] [9] [25] [24] [43] [30] [31] [32]. This type of AP only supports parallel search operation to locate data records by the content and relies on external ALUs or processors to perform arithmetic computations. The computation parallelism is then limited by the number of ALUs/processors. Several works [39] [31] [32] use the two-bit encoding technique to improve the search performance (latency and energy consumption), but they still have a limited searching capability (*Single-Search-Single-Pattern*), as they do not support the additional search keys proposed in this paper (Fig. 5c). 2) General-purpose AP [37] [56] [55] [35] that fully implements the execution model described in Section II to perform SIMD computation in the memory array. However, the performance of general-purpose AP is fundamentally limited by the traditional execution model that performs computation in the *Single-Search-Single-Pattern* and *Single-Search-Single-Write* manner (Section II-D). *Hyper-AP* provides a new abstract machine model and an enhanced execution model to fully address these limitations.

B. Other PIM Architectures

Several works [13] [49] [8] [38] [50] [15] [3] propose to build *domain-specific* PIM architectures by leveraging the inherent dot-product multiplication capability of the NVM-based crossbar array. These PIM architectures perform computations in the analog domain using the power-hungry and area-inefficient DAC/ADC devices. On the contrary, *Hyper-AP* is a *general-purpose* PIM architecture and does not require the inefficient DAC/ADC to perform computations. IMP [21] is a general-purpose PIM architecture that leverages the dot-product multiplication capability of RRAM crossbar array. It also performs computations in the analog domain using ADC/DAC, thus, it has a lower power efficiency than *Hyper-AP*. Moreover, it does not provide a support on flexible

data types, but only support 32-bit integer. On the contrary, *Hyper-AP* supports various data types and can achieve better performance when using the low-precision data types.

Previous works [48] [40] [33] [1] [22] [47] also propose to efficiently perform bulk bitwise logic operations in SRAM/DRAM to build PIM architectures. Due to the constraint of the memory circuits, these works can only perform simple logic operations with few inputs (e.g. 3 inputs). Therefore, these PIM architectures need to perform a large number of logic operations for complex arithmetic operations. On the contrary, *Hyper-AP* can perform both simple bitwise logic operations and complex arithmetic operations.

Several specialized PIM systems have also been developed for important domain of applications [6] [45] [62], which integrate compute units with memory by leveraging the 3D integration. Despite the radically different hardware implementation, they are all designed based on the same principle of placing monolithic compute units (CPU [19], GPU [58], ASIC [61], etc.) physically closer to monolithic memory, and thus, have not fully addressed the memory wall problem.

VIII. Conclusion

In this paper, we propose *Hyper-AP*, a parallel PIM processor with support for flexible data types, to address the limitations of the traditional AP and substantially improve the performance of associative processing. Specifically, *Hyper-AP* provides a new abstract machine model and an enhanced execution model, so computations can be performed in a *Single-Search-Multi-Pattern* and *Multi-Search-Single-Write* manner to dramatically reduce the number of search and write operations. We also provide a complete architecture/micro-architecture to implement the abstract machine model of *Hyper-AP*, which comprises two additional optimization techniques, i.e., the *logical-unified-physical-separated* array design and the low-cost and low-latency communication interface, to further improve the performance. Finally, a complete compilation framework is provided to reduce the programming complexity, and users can write C-like programs to run applications on *Hyper-AP*. Two optimizations (operation merging and operand embedding) are applied in the compilation process to further reduce the number of search and write operations needed for computations. Our evaluation results show that *Hyper-AP* achieve up to $4.1\times$, $54\times$ and $4.4\times$ improvement in throughput, power efficiency and area efficiency respectively compared with IMP, a recent RRAM-based data parallel processor. *Hyper-AP* also achieves $3.3\times$ speedup and $23.8\times$ energy reduction in evaluated kernels compared with IMP. Finally, our experiment confirms that *Hyper-AP* is more beneficial for the RRAM-based implementation compared with the CMOS-based implementation due to the substantially reduced write operations.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback to help improve the quality of the paper. We also would like to thank Lifu Zhang for the benchmark preparation.

References

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute Caches," in *2017 IEEE International Symposium on High Performance Computer Architecture*, 2017, pp. 481–492.

[2] J. Ajanovic, "PCI Express (PCIe) 3.0 Accelerator Features," *Intel Corporation*, p. 10, 2008.

[3] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy *et al.*, "PUMA: A Programmable Ultra-Efficient Memristor-based Accelerator for Machine Learning Inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 715–731.

[4] Y. Arafa, A.-H. A. Badawy, G. Chennupati, N. Santhi, and S. Eidenbenz, "Low Overhead Instruction Latency Characterization for Nvidia GPGPUs," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–8.

[5] ASU, "Predictive Technology Model (PTM)," http://ptm.asu.edu/.

[6] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-Data Processing: Insights From A MICRO-46 Workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[7] K. Batcher, "Bit-Serial Parallel Processing Systems," *Computers, IEEE Transactions on*, vol. C-31, no. 5, pp. 377–384, May 1982.

[8] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann Machine: A Hardware Accelerator for Combinatorial Optimization and Deep Learning," in *2016 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2016, pp. 1–13.

[9] M.-F. Chang, C.-C. Lin, A. Lee, C.-C. Kuo, G.-H. Yang, H.-J. Tsai, T.-F. Chen, S.-S. Sheu, P.-L. Tseng, H.-Y. Lee *et al.*, "17.5 A 3T1R Nonvolatile TCAM Using MLC ReRAM with Sub-1ns Search Time," in *2015 IEEE International Solid-State Circuits Conference—Digest of Technical Papers*. IEEE, 2015, pp. 1–3.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite For Heterogeneous Computing," in *2009 IEEE International Symposium on Workload Characterization*. Ieee, 2009, pp. 44–54.

[11] A. Chen, "A Comprehensive Crossbar Array Model With Solutions For Line Resistance And Nonlinear Device Characteristics," *IEEE Transactions on Electron Devices*, vol. 60, no. 4, pp. 1318–1326, 2013.

[12] P.-Y. Chen and S. Yu, "Compact Modeling of RRAM Devices and Its Applications in 1T1R and 1S1R Array Design," *IEEE Transactions on Electron Devices*, vol. 62, no. 12, pp. 4022–4028, 2015.

[13] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," in *43rd Annual International Symposium on Computer Architecture*. IEEE, 2016.

[14] Y. Choi, M. El-Khamy, and J. Lee, "Towards The Limit of Network Quantization," *arXiv preprint arXiv:1612.01543*, 2016.

[15] T. Chou, W. Tang, J. Botimer, and Z. Zhang, "CASCADE: Connecting RRAMs to Extend Analog Dataflow In An End-To-End In-Memory Processing Paradigm," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019.

[16] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[17] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to High-Performance Hardware on FPGAs," in *Field Programmable Logic and Applications*. IEEE, 2012, pp. 531–534.

[18] E. W. Davis, "STARAN Parallel Processor System Software," in *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, ser. AFIPS '74. New York, NY, USA: ACM, 1974, pp. 17–22. [Online]. Available: http://doi.acm.org/10.1145/1500175.1500179

[19] J. Draper, J. T. Barrett, J. Sondeen, S. Mediratta, C. W. Kang, I. Kim, and G. Daglikoca, "A Prototype Processing-In-Memory (PIM) Chip For the Data-Intensive Architecture (DIVA) System," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 1, pp. 73–84, 2005.

[20] C. C. Foster, *Content Addressable Parallel Processors*. John Wiley & Sons, Inc., 1976.

[21] D. Fujiki, S. Mahlke, and R. Das, "In-Memory Data Parallel Processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 1–14.

[22] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 100–113.

[23] L. Goux, A. Fantini, A. Redolfi, C. Chen, F. Shi, R. Degraeve, Y. Y. Chen, T. Witters, G. Groeseneken, and M. Jurczak, "Role of The Ta Scavenger Electrode in The Excellent Switching control and Reliability of A Scalable Low-Current Operated TiN/Ta_2O_5/Ta RRAM Device," in *2014 Symposium on VLSI Technology: Digest of Technical Papers*. IEEE, 2014, pp. 1–2.

[24] Q. Guo, X. Guo, Y. Bai, and E. İpek, "A Resistive TCAM Accelerator For Data-Intensive Computing," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 339–350.

[25] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "AC-DIMM: Associative Computing With STT-MRAM," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 189–200, 2013.

[26] M. Guy, "Fast Integer Square Root by Mr. Woo's Abacus Algorithm," https://web.archive.org/web/20120306040058/http://medialab.freaknet.org/martin/src/sqrt/sqrt.c.

[27] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing Deep Neural Networks With Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*, 2015.

[28] T. D. Han and T. S. Abdelrahman, "Reducing Branch Divergence in GPU Programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 1–8.

[29] L.-Y. Huang, M.-F. Chang, C.-H. Chuang, C.-C. Kuo, C.-F. Chen, G.-H. Yang, H.-J. Tsai, T.-F. Chen, S.-S. Sheu, K.-L. Su *et al.*, "ReRAM-based 4T2R Nonvolatile TCAM with 7× NVM-stress Reduction, and 4× Improvement in Speed-Wordlength-Capacity for Normally-Off Instant-On Filter-based Search Engines Used in Big-Data Processing," in *Symposium on VLSI Circuits Digest of Technical Papers*. IEEE, 2014.

[30] W. Huangfu, S. Li, X. Hu, and Y. Xie, "RADAR: A 3D-ReRAM Based DNA Alignment Accelerator Architecture," in *ACM/ESDA/IEEE Design Automation Conference*. IEEE, 2018, pp. 1–6.

[31] M. Imani, S. Patil, and T. Š. Rosing, "Approximate Computing Using Multiple-Access Single-Charge Associative Memory," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 3, 2016.

[32] ——, "MASC: Ultra-Low Energy Multiple-Access Single-Charge TCAM For Approximate Computing," in *2016 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2016, pp. 373–378.

[33] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-In-Memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[34] S. H. Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, "3D-Stackable Crossbar Resistive Memory Based on Field Assisted Superlinear Threshold (FAST) Selector," in *2014 IEEE International Electron Devices Meeting*. IEEE, 2014, pp. 6–7.

[35] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A Resistive CAM Processing-In-Storage Architecture for DNA Sequence Alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.

[36] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Shimakawa *et al.*, "An 8Mb Multi-Layered Cross-Point ReRAM Macro With 443MB/s Write Throughput," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 178–185, 2012.

[37] S. Khoram, Y. Zha, and J. Li, "An Alternative Analytical Approach to Associative Processing," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 113–116, 2018.

[38] M. Le Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tuma, C. Bekas, A. Curioni, and E. Eleftheriou, "Mixed-Precision In-Memory Computing," *Nature Electronics*, vol. 1, no. 4, p. 246, 2018.

[39] J. Li, R. K. Montoye, M. Ishii, and L. Chang, "1Mb 0.41μm^2 2T-2R Cell Nonvolatile TCAM With Two-Bit Encoding and Clocked Self-Referenced Sensing," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 4, pp. 896–907, 2013.

[40] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "DRISA: A Dram-based Reconfigurable In-Situ Accelerator," in *50th Annual International Symposium on Microarchitecture*. IEEE, 2017.

[41] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 2, pp. 1–30, 2014.

[42] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and Sequential Mapping With Priority Cuts," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2007, pp. 354–361.

[43] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar, "Resistive GP-SIMD Processing-In-Memory," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 4, p. 57, 2016.

[44] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri, "ASC: An Associative-Computing Paradigm," *Computer*, vol. 27, no. 11, pp. 19–25, 1994.

[45] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, "NDC: Analyzing the Impact of 3D-stacked Memory+Logic Devices on MapReduce Workloads," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2014, pp. 190–200.

[46] Quinapalus, "Calculate exp() and log() Without Multiplications," https://www.quinapalus.com/efunc.html.

[47] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch *et al.*, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 185–197.

[48] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 273–287.

[49] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 14–26.

[50] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating Graph Processing Using ReRAM," in *2018 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2018.

[51] M. Vault, "The Definitive Higher Math Guide on Long Division and Its Variants—for Integers," https://mathvault.ca/long-division/.

[52] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 29.

[53] H.-S. P. Wong *et al.*, "Metal—Oxide RRAM," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[54] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[55] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi, "A Two-Dimensional Associative Processor," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 26, no. 9, pp. 1659–1670, 2018.

[56] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive Associative Processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2014.

[57] Y. Zha and J. Li, "Liquid Silicon: A Data-Centric Reconfigurable Architecture Enabled by RRAM Technology," in *International Symposium on Field-Programmable Gate Arrays*. ACM/SIGDA, 2018, pp. 51–60.

[58] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing In Memory," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014.

[59] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental Network Quantization: Towards Lossless CNNs With Low-Precision Weights," *arXiv preprint arXiv:1702.03044*, 2017.

[60] K. Zhou, X. Xue, J. Yang, X. Xu, H. Lv, M. Wang, W. Liu, X. Zeng, S. S. Chung, J. Li *et al.*, "Nonvolatile Crossbar 2D2R TCAM with Cell Size of 16.3F^2 and K-means Clustering for Power Reduction," in *Asian Solid-State Circuits Conference*. IEEE, 2018, pp. 135–138.

[61] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3D-Stacked Logic-In-Memory Accelerator For Application-Specific Data Intensive Computing," in *2013 IEEE International 3D Systems Integration Conference*. IEEE, 2013, pp. 1–7.

[62] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating Sparse Matrix-Matrix Multiplication With 3D-Stacked Logic-In-Memory Hardware," in *2013 IEEE High Performance Extreme Computing Conference*. IEEE, 2013, pp. 1–6.